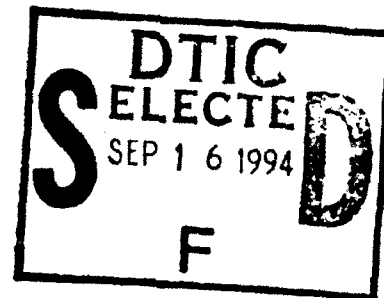


COMPUTER COMMAND AND CONTROL COMPANY

2300 CHESTNUT STREET, SUITE 230 • PHILADELPHIA, PA 19103
215-854-0555 FAX: 215-854-0665

1

AD-A284 409



FINAL REPORT

**REENGINEERING DOD-STD-2167A
REQUIREMENTS SPECIFICATIONS**

**CONTRACT N00014-92-C-0242
OFFICE OF NAVAL RESEARCH
ATTN: ELIZABETH WALD
CODE ONR-311
DEPARTMENT OF THE NAVY
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5000**

AUGUST 1994

This document has been approved
for public release and sale; its
distribution is unlimited.

94-29630



5026

DTIC QUALITY INSPECTED 5

94 0 7 0 0 3 7

TABLE OF CONTENTS

1.	Introduction	1
2.	Input: The Avionics Software	2
3.	Output: Software Unit Map and Software Units	3
4.	A Guide to Perusing the Outputs	5
Appendix I	Project Publications	
Appendix II	Examples of Documentation of Software Units (For Top Two Software Units of Avionics)	

LIST OF FIGURES

Figure 1:	Avionics Catalog Report.	6
Figure 2:	Avionics Software Unit Mapping Summary.	9
Figure 3:	Avionics Software Unit Map.	11
Figure 4:	Icons, Edges and Legends.	12

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification _____	
By _____	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

1. INTRODUCTION

This is the Final Report under contract N00014-92-C-0242. This contract covered work in two steps:

1. Design of the automatic system for generating DoD-STD-2167A Software Specifications from the respective software Ada code.
2. Implementation of this system.

The first step was concluded and documented in a technical report titled, "Automatic Reverse Engineering of Software to Confirm/Update Requirements Specifications", June 1993.

This report describes the second step, of implementing the automatic system and using it to process the software of an Avionics software system.

The report describes the input Avionics software (Section 2), the output of generated documents of the Avionics software (Section 3) and a guide to perusing the produced documentation (Section 4).

The documentation that is produced consists of the following:

1. A catalog report listing the produced files of Software Units (Figure 1).
2. The Software Unit Map Summary (Figure 2). This is a hierarchically ordered list of all the Software Units in the architecture of the Avionics software.
3. The Avionics Software Unit Map. This is a hierarchical tree depicting graphically the Software Unit Avionics architecture (Figure 3.)
4. For each software unit, there are:
 - a. Comments included in the code.
 - b. A graphic representation and the respective Ada code. The icons in the graphic representation are shown in Figure 4.
 - c. A listing of the Ada code of the software units. The documentation for the top two software units in the architecture hierarchy are shown in

Appendix II. (A full documentation is available but not included in this report because of its length).

The work on this contract also resulted in the following publication enclosed in Appendix I,

1. Ahrens, E. Lock and N. Prywes, "The Synthesis of Software Artifacts with Implementation: Re-creating Requirements Specifications", 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop (CESAW 1994), Washington, DC, July 1994.
2. J. Ahrens, E. Lock and N. Prywes, "Technology for Re-Creating Documentation", 4th Reengineering Forum, Victoria, BC, Canada, September 1994.

2. Input: The Avionics Software

The *Avionics* demonstration example is a subset of a helicopter simulation system originally programmed in Ada. This subset specifically deals with the fly-to-point operation of a helicopter navigation sub-system. X and Y coordinates are input and a clock pulse relays real-time event input for calculation of speed, altitude, trajectory, and distance. The *Avionics* software is comprised of a series of Ada tasks: one for the clock, one for the simulation, two for input and output, two for message mailbox handling, and one controller for activating and synchronizing the other tasks; in total, seven tasks facilitate the actions of the overall system. In this demonstration example of the *Avionics* system, there are 842 lines of source Ada code.

3. Output: Software Unit Map and Software Units

The purpose of the *Avionics* documentation shown below is to illustrate the various components of the re-engineering transformation process. The system-wide "road-map" is the *Catalog Report* (figure 1); this report indicates a particular transformation component and shows its position within the hierarchy of the *transformation unit*. A transformation unit is the total code submitted at one time for processing. During the transformation, criteria for *software unit* partitioning are specified to aid in software understanding when the visualization tool is utilized. The *Software Unit Mapping Report* (figure 2) identifies the partitioning and encapsulation criteria, the number of resultant *software units*, the name of each unit, and each unit's *tree code* which identifies its position in the hierarchical breakdown of the transformation unit.

In the transformation process of re-engineering this code, 25 *in-the-small* (ITS) software units were created. (*In-the-small* entities are Ada bodies: package, procedure, function, task, generic, etc.). The partitioning and encapsulation criteria produced 34 *software units*, which are *in-the-large* (ITL) entities (i.e. containing no Ada bodies). In total there are 59 software units. Each of

these transformation entities is identified within the documentation, and their corresponding Ada and graphical views are presented.

The partitioning and encapsulation phase of the transformation process creates a natural hierarchy among the partitioned *software units*. This hierarchy is signified textually via a numerical indicator called a *tree code*. Much like sub-sections of a paper, the dot (.) notation in the *tree code* indicates descendant/ancestor relationships. The breakdown and representation of the overall resulting hierarchy is textually represented via the *Software Unit Mapping Report*. In addition to this report there is a graphical view which illustrates the hierarchical relationship via a standard ER style representation. This *Software Unit Map* view is shown in Figure 3 below. In order to understand the graphical views, Figure 4 shows the icons and edges along with their symbolic meanings.

In regard to Figure 3, the *Avionics Software Unit Map*, the partitioning and organization is explained in the following. At the highest level there is the *system* node AVIONICS, its tree code is 1 signifying it as the root node. That system can be logically partitioned into three child systems: HELI_MAILBOXING (1.1), HELI_PROCEDURES (1.2), and HELICOPTER (1.3). These three partitions encapsulate the functionality of mailboxing tasks in HELI_MAILBOXING (1.1), support procedures in HELI_PROCEDURES (1.2), and the main procedure, support procedure tasking shells and controller task in HELICOPTER (1.3). By creating this type of partitioning, functions logically associated to each of these child systems can be easily added or maintained. For further definition of each of these three child systems, they in turn are partitioned into even more specific functional systems. Each is described as a separate section below.

1.1 **HELI_MAILBOXING** is partitioned into two child systems: HELI_MBX_WITH_AND_USE (1.1.1) contains all the *context* nodes (i.e. Ada "with" and "use") for HELI_MAILBOXING (1.1); HELI_MBX_PACKAGE (1.1.2) contains two child systems which represent the main mailboxing tasks. EXAMPLE_HELI_IO_MBX (1.1.2.1) maintains the mailbox for all I/O messages; EXAMPLE_HELI_SIM_MBX (1.1.2.2) maintains the mailbox for all messages to and from the simulation module. If additional mailboxing tasks were required, or the existent ones needed updating, this organization of partitioning allows for clear and understandable functional delineation.

1.2 **HELI_PROCEDURES** also is partitioned into two child systems. HELI_PROC_WITH_AND_USE (1.2.1) contains all the context nodes for HELI_PROCEDURES (1.2). The system HELI_PROC_PACKAGE (1.2.2) represents a package of all the main supporting procedures.

CLOCK (1.2.2.1) is the system representing the clocking procedure for real-time event pulse computation. It has two systems representing: the function GET_MAX (1.2.2.1.1) which relays the number of tasks in its execution diameter; and the procedure MBX_WRITE (1.2.2.1.2) which sends the clock pulse to the simulation mailbox.

HELI_IO_PROCS (1.2.2.2) is a system provided for logically grouping the I/O procedures. There are two I/O procedures which deal with operator input (via a keyboard) which requires

parsing, and informatory output (via a display). Therefore, two child systems appear. DISPLAY (1.2.2.2.1) handles all output to the screen. DISPLAY receives data from a mailbox through its child procedure MBX_READ (1.2.2.2.1.2). DISPLAY's other child, GET_MAX (1.2.2.2.1.1), relays the number of tasks in its execution diameter. The input system procedure is PARSER (1.2.2.2.2); it is responsible for obtaining and parsing user input; it then sends information to either the simulation module or to the display module via a mailbox. The PARSER child system MBX_WRITE (1.2.2.2.2.2) handles the output going to the mailbox. The other child, GET_MAX (1.2.2.2.2.1), relays the number of tasks in its execution diameter.

Finally, the system SIMULATE (1.2.2.3) represents the main simulation procedure. It is this module which calculates fly-to-point parameters to carry out the helicopter simulation. SIMULATE receives and sends data via a mailbox. Therefore, it has two child systems to handle these actions: MBX_READ (1.2.2.3.2) for input messages from CLOCK and PARSER, and MBX_WRITE (1.2.2.3.3) for output messages to DISPLAY. GET_MAX (1.2.2.3.1) relays the number of tasks in its execution diameter.

By partitioning the main supporting procedures in this manner, adding a new system like RADAR becomes very straight-forward. It is clear within the above partitioning that a system node such as RADAR could be added with a tree code of 1.2.2.4; a brother node to SIMULATE (1.2.2.3).

1.3 HELICOPTER is the third child system of AVIONICS. Its child HELICOPTER_WITH_AND_USE (1.3.1) is a system containing the context nodes for HELICOPTER. The system HELI (1.3.2) breaks down into four child systems which in turn control the entire simulation. CLOCK_S (1.3.2.1) is the task shell for the CLOCK procedure. HELICOPTER_IO_TASKS (1.3.2.2) logically groups two children associated with I/O: DISPLAY_S (1.3.2.2.1) is the task shell for the DISPLAY procedure, and PARSER_S (1.3.2.2.2) is the task shell for the PARSER procedure. SIMULATE_S (1.3.2.3) is the task shell for the SIMULATE procedure. The system CONTROLLER (1.3.2.4) is a task responsible for starting and coordinating all of the other tasks. It utilizes its child procedure system INDX (1.3.2.4.1) to start the tasks in the appropriate order. The child function system ALL_DONE (1.3.2.4.2) informs the CONTROLLER when the tasks have terminated, thereby allowing it to terminate the simulation.

The SRE environment is flexible enough to allow the user to modify this partitioning at any time during the Software Understanding process. Through the addition of system nodes or a change in partitioning criteria, the view of software units can be produced to satisfy any level of system delineation for better understanding. The above partitioning example provided a clearer understanding of the Avionics source Ada which was not organized in such a functional manner. By using system nodes and adding criteria for package, task, procedure, and function partitioning the above mapping was produced.

4. A Guide to Perusing the SRE Outputs

Throughout the SRE re-engineering transformation process, reports are produced which aid in understanding. As mentioned earlier, the SRE environment maintains a *catalog* of *transformation units*. Each *transformation unit* has many sub-entities related to it: original source code files, ESL files, software units, in-the-small units, report files, possibly generated Ada and eventually compilation make files. The *catalog* identifies each of these entities by a name and points to its physical disk location. The *tree code* hierarchy mentioned above is also stored in the *catalog* so that the user may have "quick" knowledge of the *transformation unit's* overall breakdown in hierarchy.

During the *Software Understanding* phase, which utilizes DEC Design, the user may perform a number of retrievals which help to more fully understand a piece of the resultant transformation. These retrievals may be textual, such as all the comments within a software unit, or graphical. This "book" is a compilation of information for each of the resultant sub-entities of the *Avionics transformation unit*. The *Catalog Report* shows each entity entry and the *Software Unit Mapping Report* summarizes the naming and hierarchy of the *software units*. For each *software unit* and each *in-the-small unit* there can be a number of retrievals. Within this report appear 3 basic retrievals: comments for each unit, the graphical view of the unit showing only scope edges, and the generated Ada for the unit. As such, each unit is arranged according to its position in the hierarchical breakdown by *tree code* reference. The *software units* appear first and are then followed by the *in-the-small units*. Please refer to Figure 4 for interpreting the SRE icons and edges when regarding a graphical view.

 Computer Command and Control Company SRE Catalog Report Generator

 CATALOG FILE REPORT FOR: DATESDISK:[SRE_TEST.RT]CATALOG.DAT

catalog report created on 5-4-1994 at 15:00:20

This catalog contains the following Transformation Unit(s):
 avionics

Transformation Unit: avionics

Source Language Files: language (Ada)
 kiwi\$dub0:[sre_test.rt]avionics_test.ada
 ESL Tuple Tree File: avionics (ESL Tuple Tree)
 kiwi\$dub0:[sre_test.rt]avionics.tpl
 ESL Symbol Tree File: avionics (ESL Symbol Tree)
 kiwi\$dub0:[sre_test.rt]avionics.sym
 Unit ESL Files:
 In the Large: avionics (ESL In the Large)
 kiwi\$dub0:[sre_test.rt]avionics.lrg
 Unit Mapping File: avionics (Software Unit Map)
 kiwi\$dub0:avionics_e_unit_map.
 Unit Mapping Report: avionics (Software Unit Map Report)
 kiwi\$dub0:avionics_e_unit_map.txt
 Software Unit Files: 1 AVIONICS (SYSTEM)
 kiwi\$dub0:[sre_test.rt]AVIONICS.1
 1.1 HELI_MAILBOXING (SYSTEM)
 kiwi\$dub0:[sre_test.rt]HELI_MAILBOXING.1_1
 1.1.1 HELI_MBX_WITH_AND_USE (SYSTEM)
 kiwi\$dub0:[sre_test.rt]HELI_MBX_WITH_AND_USE.1_1_1
 1.1.2 HELI_MBX_PACKAGE (PACK_BODY)
 kiwi\$dub0:[sre_test.rt]HELI_MBX_PACKAGE.1_1_2
 1.1.2.1 EXAMPLE_HELI_IO_MBX (TASK_BODY)
 kiwi\$dub0:[sre_test.rt]EXAMPLE_HELI_IO_MBX.1_1_2_1
 1.1.2.2 EXAMPLE_HELI_SIM_MBX (TASK_BODY)
 kiwi\$dub0:[sre_test.rt]EXAMPLE_HELI_SIM_MBX.1_1_2_2
 1.2 HELI_PROCEDURES (SYSTEM)
 kiwi\$dub0:[sre_test.rt]HELI_PROCEDURES.1_2
 1.2.1 HELI_PROC_WITH_AND_USE (SYSTEM)
 kiwi\$dub0:[sre_test.rt]HELI_PROC_WITH_AND_USE.1_2_1
 1.2.2 HELI_PROC_PACKAGE (PACK_BODY)
 kiwi\$dub0:[sre_test.rt]HELI_PROC_PACKAGE.1_2_2
 1.2.2.1 CLOCK (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]CLOCK.1_2_2_1
 1.2.2.1.1 GET_MAX (FCN_BODY)
 kiwi\$dub0:[sre_test.rt]GET_MAX.1_2_2_1_1
 1.2.2.1.2 MBX_WRITE (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]MBX_WRITE.1_2_2_1_2
 1.2.2.2 HELI_IO_PROCS (SYSTEM)
 kiwi\$dub0:[sre_test.rt]HELI_IO_PROCS.1_2_2_2
 1.2.2.2.1 DISPLAY (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]DISPLAY.1_2_2_2_1
 1.2.2.2.1.1 GET_MAX (FCN_BODY)
 kiwi\$dub0:[sre_test.rt]GET_MAX.1_2_2_2_1_1
 1.2.2.2.1.2 MBX_READ (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]MBX_READ.1_2_2_2_1_2
 1.2.2.2.2 PARSER (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]PARSER.1_2_2_2_2
 1.2.2.2.2.1 GET_MAX (FCN_BODY)
 kiwi\$dub0:[sre_test.rt]GET_MAX.1_2_2_2_2_1
 1.2.2.2.2.2 MBX_WRITE (PROC_BODY)
 kiwi\$dub0:[sre_test.rt]MBX_WRITE.1_2_2_2_2_2

Figure 1: Avionics Catalog Report.


```

1.2.2.3 SIMULATE (PROC_BODY)
  kiwi$dub0:[sre_test.rt]SIMULATE.1_2_2_3
1.2.2.3.1 GET_MAX (FCN_BODY)
  kiwi$dub0:[sre_test.rt]GET_MAX.1_2_2_3_1
1.2.2.3.2 MBX_READ (PROC_BODY)
  kiwi$dub0:[sre_test.rt]MBX_READ.1_2_2_3_2
1.2.2.3.3 MBX_WRITE (PROC_BODY)
  kiwi$dub0:[sre_test.rt]MBX_WRITE.1_2_2_3_3
1.3 HELICOPTER (SYSTEM)
  kiwi$dub0:[sre_test.rt]HELICOPTER.1_3
1.3.1 HELICOPTER_WITH_AND_USE (SYSTEM)
  kiwi$dub0:[sre_test.rt]HELICOPTER_WITH_AND_USE.1_3_1
1.3.2 HELI (PROC_BODY)
  kiwi$dub0:[sre_test.rt]HELI.1_3_2
1.3.2.1 CLOCK_S (TASK_BODY)
  kiwi$dub0:[sre_test.rt]CLOCK_S.1_3_2_1
1.3.2.2 HELICOPTER_IO_TASKS (SYSTEM)
  kiwi$dub0:[sre_test.rt]HELICOPTER_IO_TASKS.1_3_2_2
1.3.2.2.1 DISPLAY_S (TASK_BODY)
  kiwi$dub0:[sre_test.rt]DISPLAY_S.1_3_2_2_1
1.3.2.2.2 PARSER_S (TASK_BODY)
  kiwi$dub0:[sre_test.rt]PARSER_S.1_3_2_2_2
1.3.2.3 SIMULATE_S (TASK_BODY)
  kiwi$dub0:[sre_test.rt]SIMULATE_S.1_3_2_3
1.3.2.4 CONTROLLER (TASK_BODY)
  kiwi$dub0:[sre_test.rt]CONTROLLER.1_3_2_4
1.3.2.4.1 INDX (FCN_BODY)
  kiwi$dub0:[sre_test.rt]INDX.1_3_2_4_1
1.3.2.4.2 ALL_DONE (FCN_BODY)
  kiwi$dub0:[sre_test.rt]ALL_DONE.1_3_2_4_2
In the Small:
1.1.2 HELI_MBX_PACKAGE (ESL in the small)
  kiwi$dub0:[sre_test.rt]HELI_MBX_PACKAGE.SML
1.1.2.1 EXAMPLE_HELI_IO_MBX (ESL in the small)
  kiwi$dub0:[sre_test.rt]EXAMPLE_HELI_IO_MBX.SML
1.1.2.2 EXAMPLE_HELI_SIM_MBX (ESL in the small)
  kiwi$dub0:[sre_test.rt]EXAMPLE_HELI_SIM_MBX.SML
1.2.2 HELI_PROC_PACKAGE (ESL in the small)
  kiwi$dub0:[sre_test.rt]HELI_PROC_PACKAGE.SML
1.2.2.1 CLOCK (ESL in the small)
  kiwi$dub0:[sre_test.rt]CLOCK.SML
1.2.2.1.1 GET_MAX (ESL in the small)
  kiwi$dub0:[sre_test.rt]GET_MAX.SML
1.2.2.1.2 MBX_WRITE (ESL in the small)
  kiwi$dub0:[sre_test.rt]MBX_WRITE.SML
1.2.2.2.1 DISPLAY (ESL in the small)
  kiwi$dub0:[sre_test.rt]DISPLAY.SML
1.2.2.2.1.1 GET_MAX_1 (ESL in the small)
  kiwi$dub0:[sre_test.rt]GET_MAX_1.SML
1.2.2.2.1.2 MBX_READ (ESL in the small)
  kiwi$dub0:[sre_test.rt]MBX_READ.SML
1.2.2.2.2 PARSER (ESL in the small)
  kiwi$dub0:[sre_test.rt]PARSER.SML
1.2.2.2.2.1 GET_MAX_2 (ESL in the small)
  kiwi$dub0:[sre_test.rt]GET_MAX_2.SML
1.2.2.2.2.2 MBX_WRITE_1 (ESL in the small)
  kiwi$dub0:[sre_test.rt]MBX_WRITE_1.SML
1.2.2.3 SIMULATE (ESL in the small)
  kiwi$dub0:[sre_test.rt]SIMULATE.SML
1.2.2.3.1 GET_MAX_3 (ESL in the small)
  kiwi$dub0:[sre_test.rt]GET_MAX_3.SML
1.2.2.3.2 MBX_READ_1 (ESL in the small)
  kiwi$dub0:[sre_test.rt]MBX_READ_1.SML
1.2.2.3.3 MBX_WRITE_2 (ESL in the small)
  kiwi$dub0:[sre_test.rt]MBX_WRITE_2.SML

```

Figure 1: Avionics Catalog Report (Continued).

```
1.3.2 HELI (ESL in the small)
    kiwi$dub0:[sre_test.rt]HELI.SML
1.3.2.1 CLOCK_S (ESL in the small)
    kiwi$dub0:[sre_test.rt]CLOCK_S.SML
1.3.2.2.1 DISPLAY_S (ESL in the small)
    kiwi$dub0:[sre_test.rt]DISPLAY_S.SML
1.3.2.2.2 PARSER_S (ESL in the small)
    kiwi$dub0:[sre_test.rt]PARSER_S.SML
1.3.2.3 SIMULATE_S (ESL in the small)
    kiwi$dub0:[sre_test.rt]SIMULATE_S.SML
1.3.2.4 CONTROLLER (ESL in the small)
    kiwi$dub0:[sre_test.rt]CONTROLLER.SML
1.3.2.4.1 INDX (ESL in the small)
    kiwi$dub0:[sre_test.rt]INDX.SML
1.3.2.4.2 ALL_DONE (ESL in the small)
    kiwi$dub0:[sre_test.rt]ALL_DONE.SML
Compilation Makefile: avionics (Compilation Order)
                        file: not available
End Transformation Unit
```

Figure 1: Avionics Catalog Report (continued).

PARTITION and ENCAPSULATION SUMMARY			
PROJECT:	avionics		
DATE:	Wed Apr 27 16:28:13 1994		
ENCAPSULATION:	INCLUDED		
PARTITION CRITERIA:			
	SYSTEM		
	PROGRAM_FILE		
	PACK_BODY		
	TASK_BODY		
	PROC_BODY		
	FCN_BODY		
TOTAL UNITS:	34		
NOTE: The hierarchy of Software Units is represented by the tree codes.			

1	AVIONICS	(SYSTEM)	:50 nodes
	AVIONICS.1		
1.1	HELI_MAILBOXING	(SYSTEM)	:24 nodes
	HELI_MAILBOXING.1.1		
1.1.1	HELI_MBX_WITH_AND_USE	(SYSTEM)	:8 nodes
	HELI_MBX_WITH_AND_USE.1.1.1		
1.1.1.1	HELI_MBX_PACKAGE	(PACK_BODY)	:9 nodes
	HELI_MBX_PACKAGE.1.1.1.1		
1.1.1.1.1	EXAMPLE_HELI_IO_MBX	(TASK_BODY)	:2 nodes
	EXAMPLE_HELI_IO_MBX.1.1.1.1.1		
1.1.1.1.1.1	EXAMPLE_HELI_SIM_MBX	(TASK_BODY)	:2 nodes
	EXAMPLE_HELI_SIM_MBX.1.1.1.1.1.1		
1.1.1.1.1.1.1	EXAMPLE_HELI_SIM_MBX.1.1.1.1.1.1.1		
1.2	HELI_PROCEDURES	(SYSTEM)	:45 nodes
	HELI_PROCEDURES.1.2		
1.2.1	HELI_PROC_WITH_AND_USE	(SYSTEM)	:19 nodes
	HELI_PROC_WITH_AND_USE.1.2.1		
1.2.1.1	HELI_PROC_PACKAGE	(PACK_BODY)	:17 nodes
	HELI_PROC_PACKAGE.1.2.1.1		
1.2.1.1.1	CLOCK	(PROC_BODY)	:7 nodes
	CLOCK.1.2.1.1.1		
1.2.1.1.1.1	GET_MAX	(FCN_BODY)	:2 nodes
	GET_MAX.1.2.1.1.1.1		
1.2.1.1.1.1.1	MBX_WRITE	(PROC_BODY)	:2 nodes
	MBX_WRITE.1.2.1.1.1.1.1		
1.2.1.1.1.1.1.1	MBX_WRITE.1.2.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1	HELI_IO_PROCS	(SYSTEM)	:27 nodes
	HELI_IO_PROCS.1.2.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1	HELI_IO_PROCS.1.2.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1	DISPLAY	(PROC_BODY)	:10 nodes
	DISPLAY.1.2.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1	DISPLAY.1.2.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1	GET_MAX	(FCN_BODY)	:2 nodes
	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_READ	(PROC_BODY)	:2 nodes
	MBX_READ.1.2.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_READ.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	PARSER	(PROC_BODY)	:8 nodes
	PARSER.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	PARSER.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	GET_MAX	(FCN_BODY)	:2 nodes
	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_WRITE	(PROC_BODY)	:2 nodes
	MBX_WRITE.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_WRITE.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_WRITE	(PROC_BODY)	:13 nodes
	MBX_WRITE.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	SIMULATE	(FCN_BODY)	:2 nodes
	SIMULATE.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	GET_MAX	(FCN_BODY)	:2 nodes
	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	GET_MAX.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1		
1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1	MBX_READ	(PROC_BODY)	:2 nodes

Figure 2: Avionics Software Unit Mapping Summary.

	MBX_READ.1_2_2_3_2		
1.2.2.3.3	MBX_WRITE	(PROC_BODY)	:2 nodes
	MBX_WRITE.1_2_2_3_3		
1.3	HELICOPTER	(SYSTEM)	:22 nodes
	HELICOPTER.1_3		
1.3.1	HELICOPTER_WITH_AND_USE	(SYSTEM)	:20 nodes
	HELICOPTER_WITH_AND_USE.1_3_1		
1.3.2	HELI	(PROC_BODY)	:23 nodes
	HELI.1_3_2		
1.3.2.1	CLOCK_S	(TASK_BODY)	:2 nodes
	CLOCK_S.1_3_2_1		
1.3.2.2	HELICOPTER_IO_TASKS	(SYSTEM)	:10 nodes
	HELICOPTER_IO_TASKS.1_3_2_2		
1.3.2.2.1	DISPLAY_S	(TASK_BODY)	:2 nodes
	DISPLAY_S.1_3_2_2_1		
1.3.2.2.2	PARSER_S	(TASK_BODY)	:2 nodes
	PARSER_S.1_3_2_2_2		
1.3.2.3	SIMULATE_S	(TASK_BODY)	:2 nodes
	SIMULATE_S.1_3_2_3		
1.3.2.4	CONTROLLER	(TASK_BODY)	:6 nodes
	CONTROLLER.1_3_2_4		
1.3.2.4.1	INDX	(FCN_BODY)	:2 nodes
	INDX.1_3_2_4_1		
1.3.2.4.2	ALL_DONE	(FCN_BODY)	:2 nodes
	ALL_DONE.1_3_2_4_2		

Figure 2: Avionics Software Unit Mapping Summary (continued).

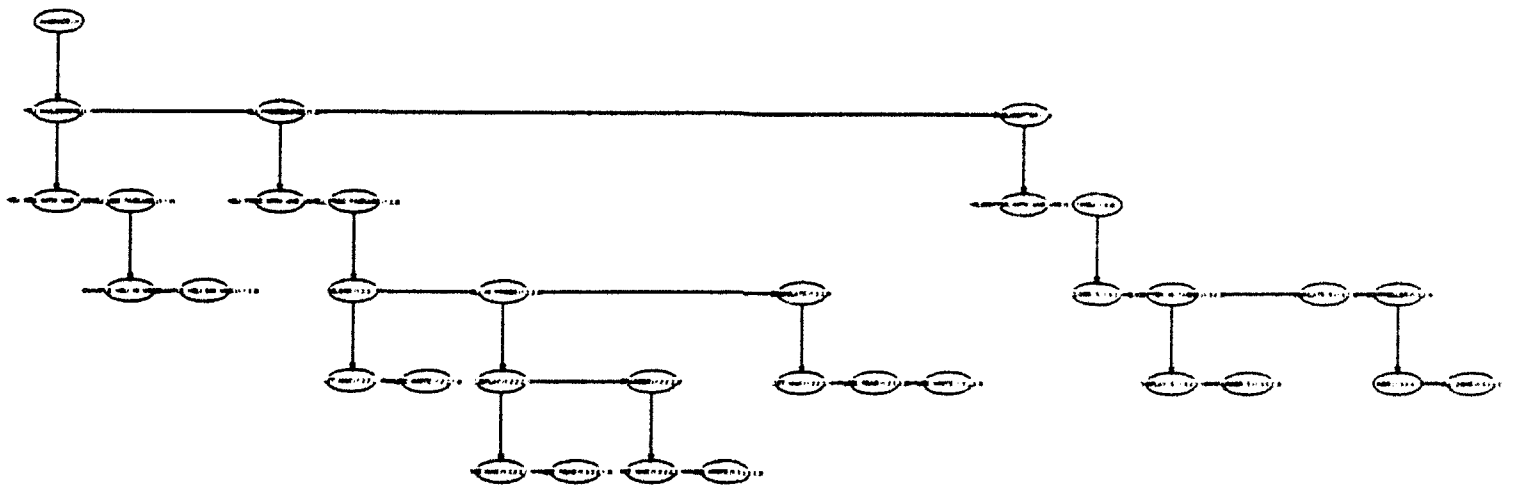


Figure 3: Avionics Software Unit Map.




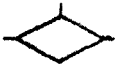












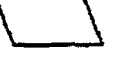









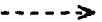



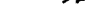
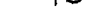
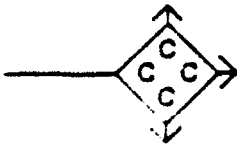
			
Package Generic	Proc. Fcn. Generic	Task Entry	Cond Stmt
			
Package Spec	Proc. Fcn. Spec	IO File	Assign Stmt
			
Package Body	Proc. Fcn. Body	Comments	Call Stmt
			
Task Type	Variable Type	Control Stmt	Message Stmt
			
Task Spec	Variables	Context Stmt	IO Stmt
			
Task Body	System	Begin etc Stmt	Loop Stmt
			
Scope Tuple (Straight)	Call Tuple	IO Tuple	Context Tuple
			
Memory Tuple	Type Tuple	Entry_Call Tuple	Schedule Tuple

Figure 4: Icons, Edges, and Legends.

APPENDIX I

PROJECT PUBLICATIONS



COMPUTER COMMAND AND CONTROL COMPANY

2300 CHESTNUT STREET, SUITE 230 • PHILADELPHIA PA 19103
215-854-0555 FAX: 215-854-0665

The Synthesis of Software Artifacts with Implementation: Re-creating Requirements Specifications

**1994 Complex Systems Engineering
Synthesis and Assessment
Technology Workshop (CSESAW '94)**

**July 19-20, 1994
Washington, DC**

Dr. Judith Ahrens[†], Mr. Evan Lock, and Dr. Noah S. Prywes*
Computer Command and Control Company
2300 Chestnut Street, Suite 230
Philadelphia, PA 19103
Tel: 215-854-0555, Fax: 215-854-0665
Email: lock@cccc.com

[†]Also with Drexel University

*Also with University of Pennsylvania

Abstract

There is a strong tendency in the Department of Defense Programs to re-develop new software when legacy software can be reused at lower cost, reduced development time and higher quality based on real-life experience. The decisions for re-developing software are frequently based on inadequacy, or sometimes lack, of documentation and the difficulty of understanding the legacy software. The cost of software understanding has been estimated at 50% of the cost of reengineering.

This paper describes an approach and toolset that synthesizes automatic processing of legacy code to produce a graphical explanation of the software architecture, and to generate reliable software specifications documents in accordance with DOD-STD instructions. The need for this capability is widely recognized.

This capability has been under development for the past 3 years. It consists of integrating the Software Reengineering Environment (SRE), funded by the Naval Surface Warfare Center (NSWC), and the Software Specification Assistant (SSA), funded by the Joint Logistics Commanders-Joint Policy Coordinating Group on Computer Resource Management (JLC-CRM). The development is nearing completion and a demonstration project is planned.

1. MOTIVATION

At a recent workshop on Reengineering (Fourth Systems Reengineering Technology Workshop, February 8-11, 1994, Monterey, CA), several speakers reported that understanding legacy software accounts for 50% of the cost and time of reengineering. At this high cost, Program Managers tend to write off all or parts of the legacy software and develop new system modules or entirely new systems. Thus, there has been a widely recognized need for automating software understanding.

Another widely recognized problem is the frequent unreliability, incompleteness and sometimes total lack of software documentation. Software documentation is produced in many software development projects as the last step and tends to be short-changed. There has been no effective procedure to determine the quality of submitted documentation. The inadequacy of documentation has also prevented verification that the software provides the capabilities established in planning, specifications and contracting documents.

Programs across DOD need to be able to:

- i. Check conformance with Software Specifications in periodic Contractor reviews and upon delivery of a new system,
- ii. Update obsolete specifications,
- iii. Create specifications for undocumented software,
- iv. Understand existing software architecture (for reuse).

The proposed capabilities will have a wide ranging impact on:

- i. Reducing maintenance costs by graphically explaining the architecture and operation of the software.
- ii. Increasing system life through adding new builds incrementally based on explaining graphically the architecture and operation of the software.
- iii. Facilitating modernization through exposing the steps necessary to execute the system in a modern distributed computer communications environment.
- iv. Improving quality and usefulness of systems through facilitating verification of a system in reviews that assure conformance with the requirements, specification and contract for the system.

The next section describes the two toolsets (and their interface) that are the basis for providing the above capabilities. The third section describes how the two tools are used together to define an approach to re-create documentation. Section 4 reviews status and plans.

2. TECHNOLOGY

Two automatic tools, used in the automation of software understanding and documentation, are shown in Figure 1. They are:

- The Software Reengineering Environment (SRE) [SRE]: It has been developed under the sponsorship of the Naval Surface Warfare Center (NSWC). It incorporates software translation to Ada (from CMS-2 and in the future from FORTRAN) and the abstracting of Ada code to re-create graphically the architecture and the data and control flow.
- The Software Specification Assistant (SSA) [SSA]: It provides tools (COTS) for searching historical documents and the editing and formatting necessary for creating and updating software specifications.

Each of these environments are described further below. These descriptions provide the basis for explaining how the capabilities are synthesized as part of a cohesive approach for document re-creation.

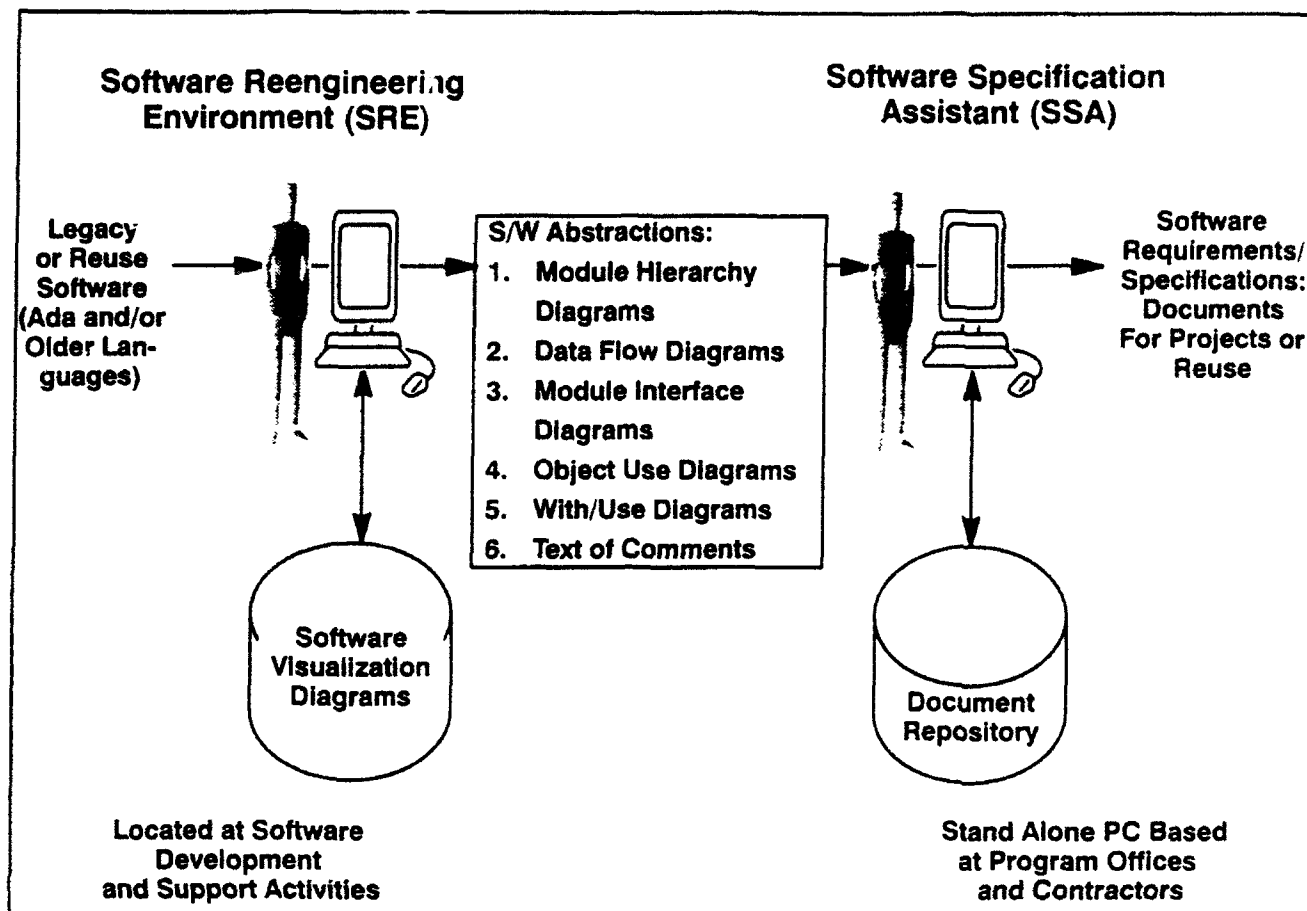


Figure 1: Process of Re-creating Software Specifications.

2.1 Software Reengineering Environment (SRE)

SRE incorporates the technologies of software translation, visualization, and understanding. SRE's architecture and capabilities are shown in Figure 2. The SRE consists of two phases, *Software Restructuring* and *Software Understanding*.

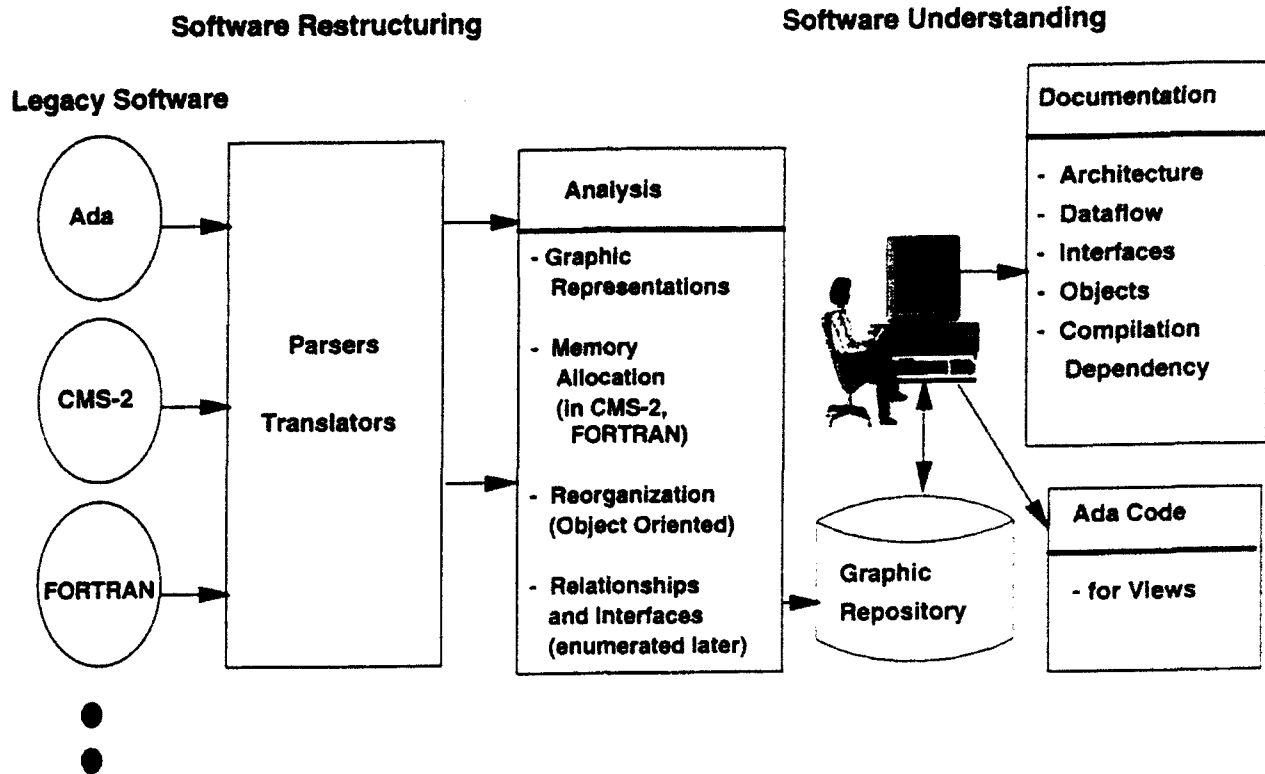


Figure 2: Architecture and Capabilities of the Software Reengineering Environment (SRE)

Software Restructuring parses and translates CMS-2, Ada and, in the future, FORTRAN code, statement by statement, into Entity-Relation-Attribute (ERA) diagrams of pseudo-Ada [PRYW]. This diagramming scheme is called Elementary Statement Language (ESL) for Ada. Next, the ESL-Ada is transformed repeatedly to obtain an Ada programming paradigm in a series of passes that achieves 100% translation to Ada. Each pass translates different aspects of the programming paradigm of the source language into the Ada programming paradigm (e.g., separating object specifications from bodies). During the translation process, a number of sets of relations among program statements are generated. The statements form nodes, and the relations form edges, in the ESL-Ada graphic diagrams.

Software Restructuring partitions the software into multi-level hierarchical software components. Software Abstraction Documents (shown in Table 1) are then generated. They describe hierarchically the architecture of these components from different perspectives. Component diagrams are stored in a graphic repository.

Abstraction Document	S/SS System/Segment	S/SDD System/Segment	SRS CSCI	IRS CSCI
Hierarchy Diagram	Par. 3.1, 3.2.3 System Architecture Diagram	Par. 4 System Architecture Diagram	Par. 3.1 CSCI External Interface Diagram	Par. 3.1. CSCI Internal Interface Diagram
Flow Diagram			Par. 3.3 CSCI Internal Interface Diagrams	
Interface Table			Par. 3.3 CSCI Internal Interface Diagrams	Par. 3.x.1 Data Element Table
Context Diagram	For Ada Compilation			
Object/Use Diagram	For Object Orientation			
Comments Text	For Capabilities			

Table 1: Mapping Software Abstraction Documents into Software Specifications.

Software Understanding (SU) consists of query and retrieval of graphic diagrams that illustrate the software from various perspectives. A graphic query language is provided for ad hoc browsing of the Software Abstraction Documents in the graphic repository. These graphs show relations between high or low level hierarchical components. This facilitates understanding of the software's architecture as well as its detailed code [PRWY3]. Facilities are being developed to make changes to the program, for debugging or program restructuring, via the graphics used for visualization. Software visualization overcomes the essential invisibility (i.e. non-physical quality) of software by representing graphically the program structure, control flow, and data. An abstract, graphical representation can facilitate a software engineer's visual perception and cognitive understanding of complex software during debugging, monitoring, and especially, program restructuring. In this way, maintenance can be performed on the reverse engineered design and/or transformed old code.

2.2 Software Specification Assistant (SSA)

SSA [SSA] was designed for project technical management. It is an integrated set of information repositories and tools for software specification of critical mission systems. It instructs and informs novice to expert staff in specifying, updating and evaluating DoD-STD data item descriptions (DIDs), including the System/Segment Specification (SSS), System/Segment Design Document (S/S, Software Requirement Specification (SRS) and Interface Requirements Specification (IRS) of DoD-STD-2167A (and similar documents on its planned successor DoD-STD-SDD) [2167A].

SSA maximizes the effectiveness of supervisory staff who are experts in the preparation of requirements specifications, and provides an automated mechanism for novices to upgrade their skills. SSA thus provides two modes of operation. In supervisory mode, using the Status Manager subsystem, the supervisor structures the requirements specification tasks and monitors progress. In activity mode, the Step-By-Step subsystem, guides a novice specifications analyst through the required work processes. SSA thus embodies much of the knowledge found in supervisory staff, enabling an organization to make efficient use of this scarce organizational resource.

SSA is composed of four customized subsystems. The function of the subsystems is as follows:

- Documentation Manager is used to create catalogs of application and reference documents in databases.
- Assignment Manager is used by a manager or supervisor to enter the work plan for staff who compose or update documents.
- Step-By-Step guides users engaged in searching documents and composing/updating Requirements and Data Items.
- Evaluate provides feedback on the completeness of the specification coverage [ARTH].

SSA also integrates the following commercial off-the-shelf software (COTS): document loading and publishing (e.g., Interleaf, MS-WORD, or Word Perfect), Search (Zyindex), CASE (depends on use by the Program Office).

The Assignment Manager subsystem enables a supervisor to create a documentation plan and assign subordinates. The process is accomplished by selecting the appropriate function from the Status Manager pulldown menu. For each project, tasks are allocated, organized and controlled through a hierarchy of three lists: Things to Do, Target Documents, and Target Document Paragraphs.

In the **Things To Do List** the supervisor enters the tasks that need to be accomplished. Examples include: work on entries from an operational requirements document's table of contents, work on items from a functional decomposition, or work on items requiring specification. For each task, the supervisor references a previously loaded document (or equivalent) that expands on the item in the Things To Do List. For example, clicking on the Things to Do list reveals a definition of its entries.

The **Target Documents List** contains the names of specification documents to be created or updated. The **Target Document Paragraph** contains the Paragraphs to be created.

At each of the three list levels, the user and the supervisor can record relevant instructions or status information such as priority of the item, problems encountered in completing the item, or sources of information used to complete the item.

After organizing the work needed to complete a plan, the supervisor assigns the work to subordinates. The subordinate will use the Step-By-Step subsystem.

The **Step-by-Step** Subsystem guides the user through the process of preparing requirements specifications. Step-by-step is an iterative process. Once a task and associated target documents are selected, the user iterates, in various combinations (even during different sessions), to search for application information and assistance, to compose data items, and to record a trace, until the selected task is completed. Then, the user will select another task from the list of Things to Do and repeat the process.

3. Approach: Interfacing SRE and SSA

The previous sections provided background on SSA and SRE. With these two capabilities available, the next question to answer is – what are the necessary abstractions that the SSA user needs to recreate specifications and can SRE produce these abstractions? The answer is that there are six basic types of abstraction/information (see Figure 1) that comprise the interface between these two tools [PRYW2].

The diagrams are created by traversing the repository for nodes with the following relations:

1. **Hierarchy Relations:** The entire repository is envisaged structured as an up-side-down tree-like hierarchy. The root unit of the tree is called a *System*. Its immediate descendants are called *Segments*. Segments can have as descendants Segments or *Computer Software Configuration Items* (CSCI). CSCIs can be object declarations, database declarations or major executable code units. CSCIs can have multiple levels of descendants called *Software Units* (SU). Software specifications document requirements/capabilities associated with each System, Segment or CSCI module in the repository.
2. **Architecture Unit Relations:** These relations are specified for each architecture unit. The interfaces are through data, transferred to or from the module or through I/O or through references.
3. **Data-Flow Relations:** These relations provide information on units that participate in a Data Flow diagram of a process accomplished by modules. The data flow relations are implemented in the programs by I/O, procedure calls or message passing.
4. **Type-Instantiation Relations:** These relations relate units that contain type (and generic) declarations with those where these declarations are used.
5. **With/Use Relations:** These relations relate units that are users of other units in a library of programs.
6. **Text of Comments** – These are related to modules through keywords.

The software abstraction process combines the above relations to produce Application Abstraction Documents (AAD). Each document is named, identifies the software being documented, specifies what kind of document it is, and specifies what are its relations to other documents. Units are either Systems, Segments, Computer Software Configuration Items (CSCI), or Software Units (SU).

The information collected during the software abstraction process is presented in six kinds of documents. All the documents focus on the high level units (systems, segments, CSCIs) of the software being abstracted. Two kinds of documents deal with the relations that exist between units:

1. **Module Hierarchy Diagrams** specify the part-of relation
2. **Context Diagrams** specify the visibility relation

Object Use Diagrams specify the subclass and instantiation relations that exist between units, and between types and data structures. Three additional kinds of diagrams describe individual units:

1. **Unit Structure Diagrams** specify the internal structure of a unit and its internal and external interactions.
2. **Interface Tables** describe in tabular form the interactions between a unit and its environment.
3. **Comment Sections** contain the comments associated to units.

No application abstraction document at present provides information on the dynamic behavior of the software being abstracted. In particular, no state diagram, event diagram, or timing diagram is

produced. At present, timing information must be obtained through existing documentation, simulation and/or instrumentation of the source code.

The above collection of graphic views is prepared by the SRE user. The diagrams are exported from SRE, catalogued by SSA's Document Manager, and loaded into SSA's search system. This can be accomplished electronically or through scanning. These diagrams and tables can be searched and portions retrieved to satisfy user interests. The SSA user progressively searches these diagrams along with prior requirements documents or other related application information to attribute capabilities and non-functional requirements to the diagrams. The diagrams can be cut and pasted directly into the appropriate sections of the requirements specification as shown in Table 1. This searching of the diagrams can also serve as the basis for exploring commonalities and variabilities of requirements for domain/application engineering.

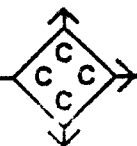
4. Status and Plans

The implementation of the converged SRE/SSA system will be completed during the summer of 1994. The plan is to follow this with a demonstration project to evaluate the system's usefulness and effectiveness. The demonstration will consist of processing existing legacy code and producing the necessary understanding and documentation. The demonstration will also compare existing software documents with those produced by the automatic system from the code. The steps in the project include:

- Step 1:** Selection of a software system to be used in the demonstration project with a participating DOD agency. The software system will have the following characteristics:
 - i. Existing software specification for later comparison with the automatically produced documentation.
 - ii. Existing Ada code of significant size (e.g., up to 1000,000 lines of code).This step will involve interviewing the DOD agency's programs and staff. The selected software will have to be available for automatic processing by CCCC.
- Step 2:** Process the selected code in the SRE. Produce the software abstraction reports discussed in Section 3.
- Step 3:** Transfer the software abstraction reports from SRE to SSA.
- Step 4:** Load the existing software specifications into SSA.
- Step 5:** Produce software specifications for the selected software.
- Step 6:** Compare the new and old specifications and produce a list of differences.

5. Bibliography

- [2167A] DOD-STD-2167A: Defense System Software Development, September 1988.
- [AHR] Ahrens, Judith, N. Prywes, and E. Lock. 4th Systems Reengineering Technology Workshop, "Maintenance Process Reengineering: Toward a New Generation of CASE Technology," Monterey, CA, February 8-10, 1994.
- [ARTH] Arthur, J. D., R. E. Nance, "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, System Research Center, Virginia Polytechnic Institute, 1987.
- [SSA] *Software Specification Assistant User's Guide: Status Manager and Step-by-Step Guide, Document Manager Guide, Evaluation Subsection Guide and Installation Guide*, delivered to the Joint Logistics Commander Computer Resource Management Sub-Group and the Office of Naval Research by Computer Command and Control Company as part of contract #N00014-91-C-0160, December 1992.
- [LOCK] Lock, E. and N. Prywes, Tri-Ada '92 Conference, "Requirements on Ada Reengineering Technology from Past, Present and Future Systems," Orlando, FL, November 16-20, 1992.
- [PRYW] Prywes, Noah, G. Ingargiola, I. Lee, and M. Lee, 4th Systems Reengineering Technology Workshop, "Reengineering Concurrent Software to Ada," Monterey, CA, February 8-10, 1994.
- [PRYW2] Prywes, N., Ingargiola, G. and Ahrens, J. "Automatic Reverse Engineering of Software to Confirm/Update Requirements Specification," Computer Command and Control Company, Contract No. N00014-92-C-0242, Philadelphia, PA, 19103, June 1993.
- [PRYW3] Prywes, N., Lee, I. "Integration of Software Specification, Reuse and Reengineering," Computer Command and Control Company, Contract No. N60921-92-C-0194, Philadelphia, PA, 19103, June 1993.
- [SRE] *Software Re-engineering Environment (SRE) Demonstration Guide*, Version 5.0. Computer Command and Control Company, Prepared Under Contract N60921-92-C-0916, White Oak Lab., Silver Spring, MD, May 6, 1994.



COMPUTER COMMAND AND CONTROL COMPANY

2300 CHESTNUT STREET, SUITE 230 • PHILADELPHIA, PA 19103
215-854-0555 FAX: 215-854-0665

Technology for Re-Creating Documentation

**4th Reengineering Forum
"Reengineering in Practice"**

**September 19-21, 1994
Victoria, BC, Canada**

Dr. Judith Ahrens[†], Mr. Evan Lock, and Dr. Noah S. Prywes*
Computer Command and Control Company
2300 Chestnut Street, Suite 230
Philadelphia, PA 19103
Tel: 215-854-0555, Fax: 215-854-0665
Email: lock@cccc.com

[†]Also with Drexel University

*Also with University of Pennsylvania

Abstract

There is a strong tendency in the Department of Defense Programs to re-develop new software when legacy software can be reused at lower cost, reduced development time and higher quality based on real-life experience. The decisions for re-developing software are frequently based on inadequacy, or sometimes lack, of documentation and the difficulty of understanding the legacy software. The cost of software understanding has been estimated at 50% of the cost of reengineering.

This paper describes an approach and toolset that synthesizes automatic processing of legacy code to produce a graphical explanation of the software architecture, and to generate reliable software specifications documents in accordance with DOD-STD instructions. The need for this capability is widely recognized.

This capability has been under development for the past 3 years. It consists of integrating the Software Reengineering Environment (SRE), funded by the Naval Surface Warfare Center (NSWC), and the Software Specification Assistant (SSA), funded by the Joint Logistics Commanders-Joint Policy Coordinating Group on Computer Resource Management (JLC-CRM). The development is nearing completion and a demonstration project is planned.

1. MOTIVATION

At a recent workshop on Reengineering (Fourth Systems Reengineering Technology Workshop, February 8-11, 1994, Monterey, CA), several speakers reported that understanding legacy software accounts for 50% of the cost and time of reengineering. At this high cost, Program Managers tend to write off all or parts of the legacy software and develop new system modules or entirely new systems. Thus, there has been a widely recognized need for automating software understanding.

Another widely recognized problem is the frequent unreliability, incompleteness and sometimes total lack of software documentation. Software documentation is produced in many software development projects as the last step and tends to be short-changed. There has been no effective procedure to determine the quality of submitted documentation. The inadequacy of documentation has also prevented verification that the software provides the capabilities established in planning, specifications and contracting documents.

Programs across DOD need to be able to:

- i. Check conformance with Software Specifications in periodic Contractor reviews and upon delivery of a new system,
- ii. Update obsolete specifications,
- iii. Create specifications for undocumented software,
- iv. Understand existing software architecture (for reuse).

The proposed capabilities will have a wide ranging impact on:

- i. Reducing maintenance costs by graphically explaining the architecture and operation of the software.
- ii. Increasing system life through adding new builds incrementally based on explaining graphically the architecture and operation of the software.
- iii. Facilitating modernization through exposing the steps necessary to execute the system in a modern distributed computer communications environment.
- iv. Improving quality and usefulness of systems through facilitating verification of a system in reviews that assure conformance with the requirements, specification and contract for the system.

The next section describes the two toolsets (and their interface) that are the basis for providing the above capabilities. The third section describes how the two tools are used together to define an approach to re-create documentation. Section 4 reviews status and plans.

2. TECHNOLOGY

Two automatic tools, used in the automation of software understanding and documentation, are used to synthesize the existing documents and code to derive requirements specifications. They are:

- The Software Reengineering Environment (SRE) [SRE]: It has been developed under the sponsorship of the Naval Surface Warfare Center (NSWC). It incorporates software translation to Ada (from CMS-2 and in the future from FORTRAN) and the abstracting of Ada code to re-create graphically the architecture and the data and control flow.
- The Software Specification Assistant (SSA) [SSA]: It provides tools (COTS) for searching historical documents and the editing and formatting necessary for creating and updating software specifications.

Each of these environments are described further below. These descriptions provide the basis for explaining how the capabilities are synthesized as part of a cohesive approach for document re-creation.

2.1 Software Reengineering Environment (SRE)

SRE incorporates the technologies of software translation, visualization, and understanding. The SRE consists of two phases, *Software Restructuring* and *Software Understanding*.

Software Restructuring parses and translates CMS-2, Ada and, in the future, FORTRAN code, statement by statement, into Entity-Relation-Attribute (ERA) diagrams of pseudo-Ada [PRYW]. This diagramming scheme is called Elementary Statement Language (ESL) for Ada. Next, the ESL-Ada is transformed repeatedly to obtain an Ada programming paradigm in a series of passes that achieves 100% translation to Ada. Each pass translates different aspects of the programming paradigm of the source language into the Ada programming paradigm (e.g., separating object specifications from bodies). During the translation process, a number of sets of relations among program statements are generated. The statements form nodes, and the relations form edges, in the ESL-Ada graphic diagrams.

Software Restructuring partitions the software into multi-level hierarchical software components. Software Abstraction Documents (see Section 3) are then generated. They describe hierarchically the architecture of these components from different perspectives. Component diagrams are stored in a graphic repository.

Software Understanding (SU) consists of query and retrieval of graphic diagrams that illustrate the software from various perspectives. A graphic query language is provided for ad hoc browsing of the Software Abstraction Documents in the graphic repository. These graphs show relations between high or low level hierarchical components. This facilitates understanding of the software's architecture as well as its detailed code [PRWY3]. Facilities are being developed to make changes to the program, for debugging or program restructuring, via the graphics used for visualization. Software visualization overcomes the essential invisibility (i.e. non-physical quality) of software by representing graphically the program structure, control flow, and data. An abstract, graphical representation can facilitate a software engineer's visual perception and cognitive understanding of complex software during debugging, monitoring, and especially, program restructuring. In this way, maintenance can be performed on the reverse engineered design and/or transformed old code.

2.2 Software Specification Assistant (SSA)

SSA [SSA] was designed for project technical management. It is an integrated set of information repositories and tools for software specification of critical mission systems. It instructs and informs novice to expert staff in specifying, updating and evaluating DoD-STD data item descriptions (DIDs), including the System/Segment Specification (SSS), System/Segment Design Document (S/S, Software Requirement Specification (SRS) and Interface Requirements Specification (IRS) of DoD-STD-2167A (and similar documents on its planned successor DoD-STD-SDD) [2167A].

SSA maximizes the effectiveness of supervisory staff who are experts in the preparation of requirements specifications, and provides an automated mechanism for novices to upgrade their skills. SSA thus provides two modes of operation. In supervisory mode, using the Status Manager subsystem, the supervisor structures the requirements specification tasks and monitors progress. In activity mode, the Step-By-Step subsystem, guides a novice specifications analyst through the required work processes. SSA thus embodies much of the knowledge found in supervisory staff, enabling an organization to make efficient use of this scarce organizational resource.

SSA is composed of four customized subsystems. The function of the subsystems is as follows:

- Documentation Manager is used to create catalogs of application and reference documents in databases.
- Assignment Manager is used by a manager or supervisor to enter the work plan for staff who compose or update documents.
- Step-By-Step guides users engaged in searching documents and composing/updating Requirements and Data Items.
- Evaluate provides feedback on the completeness of the specification coverage [ARTH].

SSA also integrates the following commercial off-the-shelf software (COTS): document loading and publishing (e.g., Interleaf, MS-WORD, or Word Perfect), Search (Zyindex), CASE (depends on use by the Program Office).

The Assignment Manager subsystem enables a supervisor to create a documentation plan and assign subordinates. The process is accomplished by selecting the appropriate function from the

Status Manager pulldown menu. For each project, tasks are allocated, organized and controlled through a hierarchy of three lists: Things to Do, Target Documents, and Target Document Paragraphs.

In the **Things To Do List** the supervisor enters the tasks that need to be accomplished. Examples include: work on entries from an operational requirements document's table of contents, work on items from a functional decomposition, or work on items requiring specification. For each task, the supervisor references a previously loaded document (or equivalent) that expands on the item in the Things To Do List. For example, clicking on the Things to Do list reveals a definition of its entries.

The **Target Documents List** contains the names of specification documents to be created or updated. The **Target Document Paragraph** contains the Paragraphs to be created.

At each of the three list levels, the user and the supervisor can record relevant instructions or status information such as priority of the item, problems encountered in completing the item, or sources of information used to complete the item.

After organizing the work needed to complete a plan, the supervisor assigns the work to subordinates. The subordinate will use the Step-By-Step subsystem.

The **Step-by-Step** Subsystem guides the user through the process of preparing requirements specifications. Step-by-step is an iterative process. Once a task and associated target documents are selected, the user iterates, in various combinations (even during different sessions), to search for application information and assistance, to compose data items, and to record a trace, until the selected task is completed. Then, the user will select another task from the list of Things to Do and repeat the process.

3. Approach: Interfacing SRE and SSA

The previous sections provided background on SSA and SRE. With these two capabilities available, the next question to answer is – what are the necessary abstractions that the SSA user needs to recreate specifications and can SRE produce these abstractions? The answer is that there are six basic types of abstraction/information that comprise the interface between these two tools [PRYW2].

The diagrams are created by traversing the repository for nodes with the following relations:

1. **Hierarchy Relations:** The entire repository is envisaged structured as an up-side-down tree-like hierarchy. The root unit of the tree is called a *System*. Its immediate descendants are called *Segments*. Segments can have as descendants Segments or *Computer Software Configuration Items* (CSCI). CSCIs can be object declarations, database declarations or major executable code units. CSCIs can have multiple levels of descendants called *Software Units* (SU). Software specifications document requirements/capabilities associated with each System, Segment or CSCI module in the repository.
2. **Architecture Unit Relations:** These relations are specified for each architecture unit. The interfaces are through data, transferred to or from the module or through I/O or through references.
3. **Data-Flow Relations:** These relations provide information on units that participate in a Data Flow diagram of a process accomplished by modules. The data

flow relations are implemented in the programs by I/O, procedure calls or message passing.

4. **Type-Instantiation Relations:** These relations relate units that contain type (and generic) declarations with those where these declarations are used.
5. **With/Use Relations:** These relations relate units that are users of other units in a library of programs.
6. **Text of Comments** – These are related to modules through keywords.

The software abstraction process combines the above relations to produce Application Abstraction Documents (AAD). Each document is named, identifies the software being documented, specifies what kind of document it is, and specifies what are its relations to other documents. Units are either Systems, Segments, Computer Software Configuration Items (CSCI), or Software Units (SU).

The information collected during the software abstraction process is presented in six kinds of documents. All the documents focus on the high level units (systems, segments, CSCIs) of the software being abstracted. Two kinds of documents deal with the relations that exist between units:

1. **Module Hierarchy Diagrams** specify the part-of relation
2. **Context Diagrams** specify the visibility relation

Object Use Diagrams specify the subclass and instantiation relations that exist between units, and between types and data structures. Three additional kinds of diagrams describe individual units:

1. **Unit Structure Diagrams** specify the internal structure of a unit and its internal and external interactions.
2. **Interface Tables** describe in tabular form the interactions between a unit and its environment.
3. **Comment Sections** contain the comments associated to units.

No application abstraction document at present provides information on the dynamic behavior of the software being abstracted. In particular, no state diagram, event diagram, or timing diagram is produced. At present, timing information must be obtained through existing documentation, simulation and/or instrumentation of the source code.

The above collection of graphic views is prepared by the SRE user. The diagrams are exported from SRE, catalogued by SSA's Document Manager, and loaded into SSA's search system. This can be accomplished electronically or through scanning. These diagrams and tables can be searched and portions retrieved to satisfy user interests. The SSA user progressively searches these diagrams along with prior requirements documents or other related application information to attribute capabilities and non-functional requirements to the diagrams. The diagrams can be cut and pasted directly into the appropriate sections of the requirements specification as shown in Table 1. This searching of the diagrams can also serve as the basis for exploring commonalities and variabilities of requirements for domain/application engineering.

4. Status and Plans

The implementation of the converged SRE/SSA system will be completed during the summer of 1994. The plan is to follow this with a demonstration project to evaluate the system's usefulness

and effectiveness. The demonstration will consist of processing existing legacy code and producing the necessary understanding and documentation. The demonstration will also compare existing software documents with those produced by the automatic system from the code. The steps in the project include:

Step 1: Selection of a software system to be used in the demonstration project with a participating DOD agency. The software system will have the following characteristics:

i. Existing software specification for later comparison with the automatically produced documentation.

ii. Existing Ada code of significant size (e.g., up to 1,000,000 lines of code).

This step will involve interviewing the DOD agency's programs and staff. The selected software will have to be available for automatic processing by CCCC.

Step 2: Process the selected code in the SRE. Produce the software abstraction reports discussed in Section 3.

Step 3: Transfer the software abstraction reports from SRE to SSA.

Step 4: Load the existing software specifications into SSA.

Step 5: Produce software specifications for the selected software.

Step 6: Compare the new and old specifications and produce a list of differences.

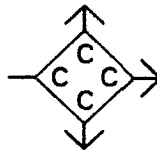
5. Bibliography

- [2167A] DOD-STD-2167A: Defense System Software Development, September 1988.
- [AHR] Ahrens, Judith, N. Prywes, and E. Lock. 4th Systems Reengineering Technology Workshop, "Maintenance Process Reengineering: Toward a New Generation of CASE Technology," Monterey, CA, February 8-10, 1994.
- [ARTH] Arthur, J. D., R. E. Nance, "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, System Research Center, Virginia Polytechnic Institute, 1987.
- [SSA] *Software Specification Assistant User's Guide: Status Manager and Step-by-Step Guide, Document Manager Guide, Evaluation Subsection Guide and Installation Guide*, delivered to the Joint Logistics Commander Computer Resource Management Sub-Group and the Office of Naval Research by Computer Command and Control Company as part of contract #N00014-91-C-0160, December 1992.
- [LOCK] Lock, E. and N. Prywes, Tri-Ada '92 Conference, "Requirements on Ada Reengineering Technology from Past, Present and Future Systems," Orlando, FL, November 16-20, 1992.
- [PRYW] Prywes, Noah, G. Ingargiola, I. Lee, and M. Lee, 4th Systems Reengineering Technology Workshop, "Reengineering Concurrent Software to Ada." Monterey, CA, February 8-10, 1994.
- [PRYW2] Prywes, N., Ingargiola, G. and Ahrens, J. "Automatic Reverse Engineering of Software to Confirm/Update Requirements Specification," Computer Command and Control Company, Contract No. N00014-92-C-0242, Philadelphia, PA, 19103, June 1993.
- [PRYW3] Prywes, N., Lee, I. "Integration of Software Specification, Reuse and Reengineering," Computer Command and Control Company, Contract No. N60921-92-C-0194, Philadelphia, PA, 19103, June 1993.
- [SRE] *Software Re-engineering Environment (SRE) Demonstration Guide*, Version 5.0, Computer Command and Control Company, Prepared Under Contract N60921-92-C-0916, White Oak Lab., Silver Spring, MD, May 6, 1994.

Technology for Re-creating Documentation

4th Reengineering Forum
September 19-21, 1994
Victoria, BC, Canada

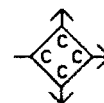
Mr. Evan Lock, President
Phone: 215-854-0555
Fax: 215-854-0665
E-mail: lock@cccc.com



Computer Command & Control Co.
2300 Chestnut St., Suite 230
Philadelphia, PA 19103

Outline

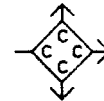
1. Motivations
2. Specification Technology (SSA)
3. Software Reengineering Environment (SRE)
4. Converging SSA with SRE



Customers/Payoff For Specification Re-creation Capability

Projects Need To Be Able To:

- Check Conformance to Software Specifications
 - Periodic Review of Contractor
 - Upon System Delivery
- Update Obsolete Specifications
- Create Specifications for Undocumented Software
- Understand Existing Software Architecture (for reuse, etc)

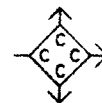


Specification Re-creation Project Description

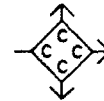
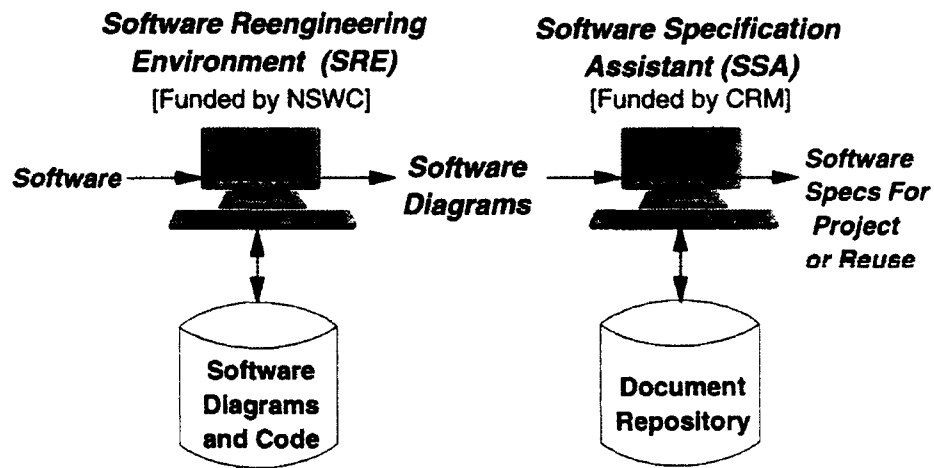
**Offer An Automated Capability to Facilitate the
Process of Re-creating Software Requirements
Specifications (SSS, SRS, IRS) From Code.**

Improve Understanding of Software To:

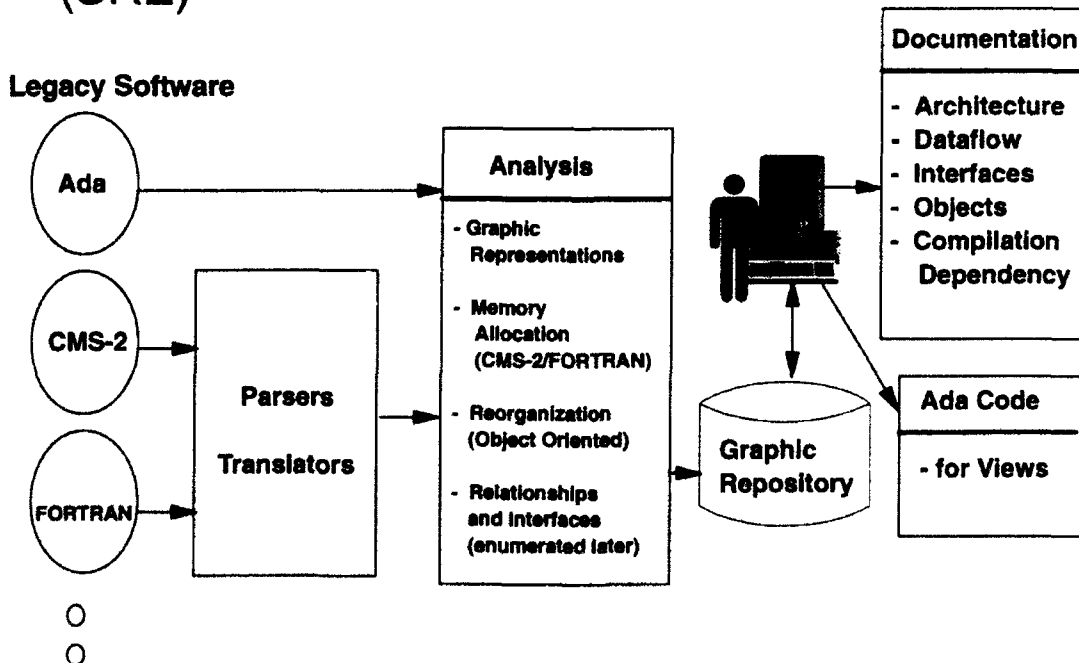
- Reduce Maintenance Costs
- Increase System Life
- Facilitate Modernization
- Better Perform Verification



Process of Re-creating Software Specifications: Convergence of SSA and SRE



Architecture and Capabilities of Software Reengineering Environment (SRE)

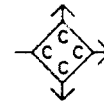


Steps In Generating The SRE Graphic Repository

**Initially Represent Entire
Software Graphically**

**Bottom-Up Build
Of Top-Down Architectural Hierarchy**

**Produce Graphic Views
For Each Architectural Unit**

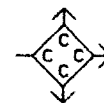


Initially Represent Entire Software Graphically

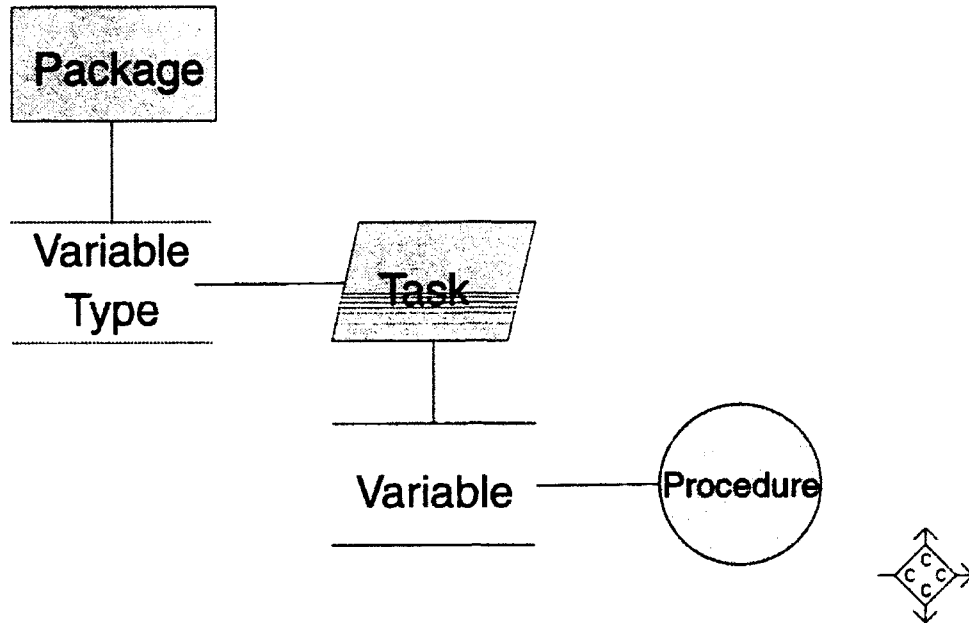
Nodes: Ada Statements

Edges:

1. Variable Reference
2. Call
3. Message
4. I/O
5. Generic/Instantiation
6. With/Use



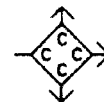
Sample "Scope" Relations



Sample "Non-Scope" Relations

For a PROCEDURE SPEC:

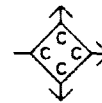
- CALLED by a procedure body
- MEMORY access
- TYPE instantiation
- CONTEXT with a package body
- CALLS a task entry
- SCHEDULEs a task spec
- I/O access to a file



Bottom-Up Build of Top-Down Architectural Hierarchy

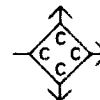
Nodes: Architectural Units
Descendent Units Represent Details
Of Parent Unit

Edges: Unit Interface (as above)

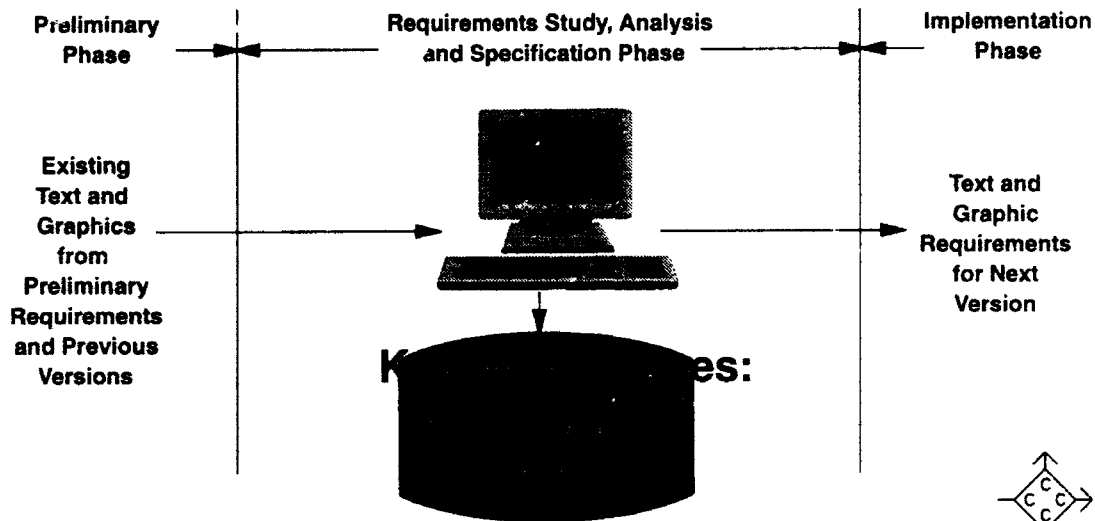


Graphical Retrievals

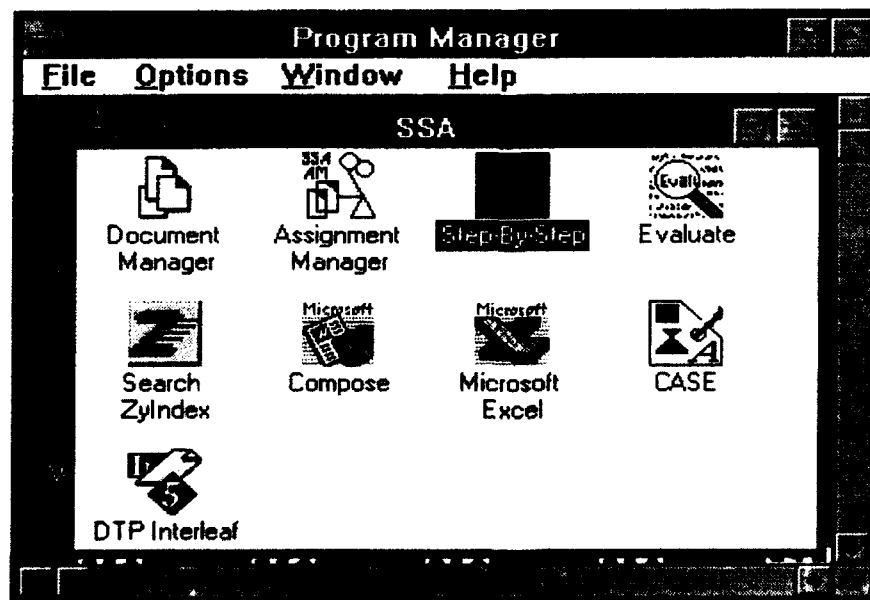
- Select a "Base View"
- Query "Base View" to Create "Sub-View"
 - Select Root Node ("Within Package X...")
 - Select Node Type ("Show Me All Procedures...")
 - Select Relations ("and Their I/O...")
 - Select Depth ("and Any Children")
- Query Sub-Views as Needed
- Save Sub-Views as Needed (for Documentation)
- Generate Ada From Any View



SSA Functionality Within the Systems Life Cycle



SSA Subsystems

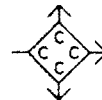


SSA Payoff in TAMPS Demo Project

Improve Project Planning and Products

- Prioritize Functions
- Evaluate Cost Benefit Tradeoffs
- Select Upgrade Path
- Reduce Risks
- Improve Quality
- Extend Life
- Reduce Costs

Enhance Expertise of Staff

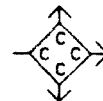


SSA Distribution to DOD

**CCCC Has Support from the JLC-CRM
to Distribute SSA to Program Offices
Across DOD.**

- Packaging
- Telephone Support
- Bug Fixes
- Training (Separate)

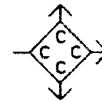
Contact for more information



Necessary Software Abstractions for Software Requirements Specifications

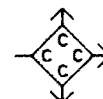
For Each Level of a System's Architecture
(Top Level, Segment, CSCI,...):

- Hierarchy Diagrams
- Data Flow Diagrams
- Interface Tables
- Context Diagrams
- Object Use Diagrams
- Comment Text



Mapping Software Abstractions Onto DOD-STD-2167A Requirements Specifications

Abstraction Document	S/SS System Segment	S/SDDSystem /Segment	SRS CSCI	IRS CSCI
Hierarchy Diagram	Par. 3.1, 3.2.3 System Architecture Diagram	Par. 4 System Architecture Diagram	Par. 3.1 CSCI External Interface Diagram	Par. 3.1 CSCI Internal Interface Diagram
Flow Diagram			Par. 3.1 CSCI Internal Interface Diagrams	
Interface Table			Par. 3.3 CSCI Internal Interface Diagrams	Par. 3.x.1 Data Element Table
Context Diagram	For Ada Compilation	For Ada Compilation	For Ada Compilation	For Ada Compilation
Object/Use Diagram	For Object Orientation	For Object Orientation	For Object Orientation	For Object Orientation
Comments Text	For Capabilities	For Capabilities	For Capabilities	For Capabilities



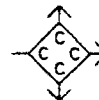
Demonstration Project Overview

Organization (MICOM) Supplies:

- Existing Software Specifications (SSS, IRS, SRS)
- Existing Code
 - Preferably Ada
 - Less Than 100K SLOC

Project Activities:

- Finalize Program Selection
- Process Code to Reverse Engineer Software Abstractions
- Assemble Specifications with Abstractions Using SSA
- Compare New Specifications with Existing
- Evaluate Process and Products



APPENDIX II

EXAMPLES OF DOCUMENTATION OF SOFTWARE UNITS (FOR TOP TWO SOFTWARE UNITS OF AVIONICS)

The Avionics system contains the pieces necessary to simulate a helicopter fly-to-point real-time routine utilizing a series of tasks.


```
-- SYSTEM : AVIONICS(1)
-- UNDCL_SYM_REC
-- PACKAGE TEXT_IO ;
-- PACKAGE CNF_PACKAGE ;
-- PACKAGE M_TYPES ;
-- PACKAGE CONV ;
-- PACKAGE MATH_OP_U ;
-- PACKAGE UNPACK_E ;
-- PACKAGE PACK_E ;
-- PACKAGE C4_SEQUENTIAL_IO ;
-- PACKAGE SEQUENTIAL_IO ;
-- PACKAGE MODEL_UFCN ;
-- PACKAGE HELI_MBX_PACKAGE ;
-- PUT_LINE : FILE_TYPE;
-- PROCEDURE CREATE ( 1 ) ;
-- MBX_WRITE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- MBX_READ : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT : FILE_TYPE;
-- PUT : FILE_TYPE;
-- PUT : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT : FILE_TYPE;
-- PUT : FILE_TYPE;
-- GET_LINE : FILE_TYPE;
-- SKIP_LINE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- Mbx_Write : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- Mbx_Read : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- PUT_LINE : FILE_TYPE;
-- Mbx_Write : FILE_TYPE;
-- PACKAGE HELI_PROC_PACKAGE ;
-- PROCEDURE CLOCK ( 1 ) ;
-- PROCEDURE DISPLAY ( 1 ) ;
-- PROCEDURE PARSER ( 1 ) ;
-- PROCEDURE SIMULATE ( 1 ) ;
-- EXAMPLE_HELI_IO_MBX_P : 3 ;
-- EXAMPLE_HELI_SIM_MBX_P : 3 ;
-- END UNDCL_SYM_REC
-- "The Avionics system contains the pieces necessary to"
-- " simulate a helicopter fly-to-point real-time routine"
-- " utilizing a series of tasks."
-- SYSTEM : HELI_MAILBOXING
-- SYSTEM : HELI_PROCEDURES
-- SYSTEM : HELICOPTER
```

The Heli_Mailboxing system contains all the componenets
related to message passing among the excuting tasks.


```
-- SYSTEM : HELI_MAILBOXING(1.1)
-- UNDCL_SYM_REC
-- PACKAGE CNF_PACKAGE ;
-- PACKAGE TEXT_IO ;
-- END UNDCL_SYM_REC
-- SYSTEM : HELI_MAILBOXING
-- "The Heli_Mailboxing system contains all the componenets"
-- " related to message passing among the excuting tasks."
-- SYSTEM : HELI_MBX_WITH_AND_USE
PACKAGE HELI_MBX_PACKAGE IS
  CONF_NAME : STRING(1..4) := "HELI" ;
  MAX_D : NATURAL:=7 ;
  TASK TYPE EXAMPLE_HELI_IO_MBX IS
    ENTRY QTEST ( STATUS : OUT INTEGER ) ;
    ENTRY SEND ( MSG : IN STRING ) ;
    ENTRY RECEIVE ( MSG : OUT STRING ) ;
  END EXAMPLE_HELI_IO_MBX ;
  TYPE A_EXAMPLE_HELI_IO_MBX IS ACCESS EXAMPLE_HELI_IO_MBX ;
  EXAMPLE_HELI_IO_MBX_P : A_EXAMPLE_HELI_IO_MBX ;
  TASK TYPE EXAMPLE_HELI_SIM_MBX IS
    ENTRY QTEST ( STATUS : OUT INTEGER ) ;
    ENTRY SEND ( MSG : IN STRING ) ;
    ENTRY RECEIVE ( MSG : OUT STRING ) ;
  END EXAMPLE_HELI_SIM_MBX ;
  TYPE A_EXAMPLE_HELI_SIM_MBX IS ACCESS EXAMPLE_HELI_SIM_MBX ;
  EXAMPLE_HELI_SIM_MBX_P : A_EXAMPLE_HELI_SIM_MBX ;
END HELI_MBX_PACKAGE ;
-- SYSTEM : HELI_MBX_PACKAGE
```

**Best
Available
Copy**